View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Elsevier - Publisher Connector

Denotational Semantics. By D.A. Schmidt. Prentice-Hall, London, 1986, Price £15.95, ISBN 0 205 10450 9.

General comments

As the author describes in the preface, denotational semantics is a methodology for giving mathematical meaning to programming languages. This book is an attempt to make this methodology more easily accessible, and to update existing texts in the area. The book emphasizes the descriptional aspects of denotational semantics but also discusses implementational concerns such as the correctness of compilers with respect to a formal semantics. While much of the text is written at a fairly informal level (and certainly there are a few sparks of wit and some idiosyncratic coining of phrases), a decent amount of mathematical detail is supplied. According to the author, the book serves as a tutorial for computing professionals and as an upper level undergraduate or graduate text. In either case, a degree of mathematical maturity and familiarity with general purpose programming languages is necessary.

The most closely related existing texts on semantics are the books by Stoy (*Denotational Semantics: The Scott–Strachey Approach to Programming Language Theory*, MIT Press, 1977) and Gordon (*The Denotational Description of Programming Languages*, Springer, 1979). Schmidt's book, published in 1986, includes some more recent material. It contains more of the relevant mathematics than does Gordon's book. I should also mention in comparison the pair of books by Milne and Strachey (*A Theory of Programming Language Semantics*, Chapman and Hall, 1976), which are much more technical and less accessible, not recommended as student texts. Tennent's book (*Principles of Programming Languages*, Prentice-Hall, 1981) emphasizes intuitive concepts rather than mathematical theory, but can be recommended to anyone with interest in the theory, design and implementation of programming languages.

The core of the book comprises the first seven chapters, together with the Introduction. The book begins with basic mathematical material; abstract syntax, structural induction and mathematical induction in Chapter 1; sets and functions in Chapter 2. Chapter 3 establishes the basics of domain theory and introduces semantic algebras and domain constructors. Chapter 4 introduces the denotational style of semantics, using as examples a binary numeral language and a simple arithmetical calculator. Chapter 5 discusses imperative languages without declarations or blocks, treating a declaration-free Pascal subset, an interactive file editor, and a dynamically typed language with input/output. Chapter 6 introduces iteration and recursion, and gives the mathematical details necessary for the standard least fixed point semantics. Chapter 7 treats block structure and declarations, introducing environments, locations and stores; the imperative language of Chapter 5 is revisited and expanded to include declarations; also included are treatments of a simple applicative language and compound data structures such as arrays and lists.

The final chapters introduce more advanced material. Chapter 8 is based on Tennent's analysis of abstraction and binding mechanisms, and includes a brief introduction to polymorphism. Chapter 9 discusses non-sequential control flow, introducing continuation semantics; examples include a FORTRAN-like **stop** command, a simple version of exception handling, a "propositional" version of PROLOG, coroutines, and unrestricted **goto**'s. Chapter 10 surveys techniques and systems for deriving compilers from denotational semantics, and touches on correctness issues, operational equivalence of program terms, and full abstraction. Chapter 11 gives a summary of Scott's inverse limit construction for the solution to a recursive domain equation; the examples treated are linear lists, self-applicative procedures, and recursive record structures. Chapter 12 introduces nondeterminism and concurrency, motivated by a parallel version of Dijkstra's language of guarded commands; a Plotkin-style resumption semantics using powerdomains, and a semantics using Milner's CCS are given.

The book ends rather abuptly at this point, and I think a final chapter surveying the state of the art and pointing to open issues and research topics would have provided a more suitable conclusion.

Exercises are provided in profusion throughout the book. This is welcome: many of them are instructive and help to clarify points made in the text. Nevertheless, the text often leaves apparently important exercises to the reader without sufficient hints (such as "finding a better solution to parameter-type enforcement" on page 185, and the proof of correctness of a stack-machine implementation in Chapter 10). Schmidt rarely provides any indication of the difficulty of problems. Instead, he says in the preface that "an hour's effort on a problem will allow the reader to make that judgement and will also aid development of intuitions about the significant problems in the area". Perhaps so, but I believe this to have been a mistake. Many students without guidance will waste time on mundane problems and come to believe that they are tackling truly significant issues. The text would also benefit from the inclusion of strategically placed fully worked proofs to illustrate the required techniques.

The references are pretty extensive. In most cases the obvious primary references are listed, but some of the secondary sources are of debatable importance. I would have preferred a better assimilation of references into the text, so that the contributions of each paper to specific parts of a chapter would be clear. Some historical perspective in the text would have helped to provide an overall feel for the subject matter.

Technical comments

Some of the author's decisions in choosing and presenting his material seem appropriate and advantageous. However, some appear to be less desirable and detract from the book's clarity and readability. I will try to summarize what I feel

to be the most prominent good and bad features. First some remarks about issues relevant to the entire book.

Schmidt says early in the book that "we formalize syntax first, because only syntactically correct programs *have* semantics" (my italics). With respect to this I would make two remarks. Firstly, it would be better to say that one need only *give* semantics to syntactically correct programs (rather than thinking of semantics as already given, it is our job as semanticists to ascribe meanings to programs). I suspect that this was the intention behind Schmidt's less pedantically chosen words. Secondly, there is a missed opportunity here to tighten up syntax by imposing static syntactic constraints to rule out certain runtime errors and thereby allow a simplified semantic treatment. At least for a simple while-loop language it would have been straightforward to prevent failure to declare or initialize variables by syntactic means, and to give an error-free semantics (since only programs satisfying the syntactic constraints need be given a semantics). The book-keeping in semantic definitions to catch and propagate errors can be tiresome to the reader and obscures the essential structure of the semantics. Admittedly, this complaint is also applicable to Stoy and Gordon, who also employ error elements extensively.

An important decision in this book is to use complete partial orders (cpo's) for semantic domains (rather than, say, complete lattices as in Stoy) and to focus on the computational relevance of concepts such as continuity. In fact, Schmidt uses "bottomless" cpo's or "predomains" in places, only insisting on the presence of a least element when necessary for the standard construction of least fixed points. By adopting a more general framework than Stoy's fix, Schmidt's treatment is less tied down to the details of a specific model.

Schmidt has obviously decided to introduce and motivate the mathematical necessities in a graduated manner, postponing questions until it is unavoidable to tackle them. Thus the required material on least fixed points of continuous functions is motivated by the need to provide a satisfactory account of while-loops, and occurs in the text along with the treatment of loops, but not before. This approach has advantages and disadvantages. While it does avoid overwhelming the reader with material above and beyond what is currently needed, it can leave the more curious reader worried about the consistency of what is being done; this is especially true with respect to recursively defined semantic domains. Such a reader would need to look ahead to Chapter 11 for reassurance.

A less desirable and pervasive notational quirk is the use of expressions like "*ninety-nine*" for the number denoted by the numeral 99. While it is certainly necessary to distinguish between numerals and numbers, i.e. between syntax and semantics, it seems rather perverse to drop the conveniences of positional notation. Most other authors use "syntactic brackets ⟦ and ⟧ to delimit syntactic terms: thus, ⟦99⟧ for the numeral, 99 for the value. Similarly, the use of "*times*" for multiplication seems rather a contrivance. In addition to this, the notation used throughout the book, especially in representing function graphs and other infinite objects, is excessively dependent on the ellipsis, a dangerously informal tool when the intention is to make definitions rigorous.

I also have a minor quibble with the interpretation Schmidt gives to his semantic notation. This relies heavily on an equational format for function definitions. Schmidt says that an equation "represents a function"; the actual function "is determined by a form of evaluation that uses substitution and simplification". I do not feel that Schmidt keeps quite the right distinctions consistently between the mathematical concept of a function (usually a graph, or set of ordered pairs), an expression in a formal or informal metalanguage which denotes such a function, and a rule or algorithm for computing the graph of a function. The equational format is part of the metalanguage for describing semantic functions, and the simplification and substitution mechanisms are rules for explaining what functions are described by which equations. In places Schmidt's usage seems confusing, apparently combining two aspects into one. I found Stoy's treatment clearer, emphasizing that the metalanguage "must be explicated using the mathematics [of domain theory]" although evaluation using conversion rules like simplification and substitution will never give the wrong answer; Stoy makes the point that attempts to evaluate (a metalanguage expression) may fail to terminate, while nevertheless the expression always represents a value in the semantic domain. Schmidt seems to treat substitution and simplification rather informally, and apparently glosses over this issue.

The next comments refer to specific sections of the book. I present them chapter-by-chapter.

There is a slight problem in Chapter 2 with the statement on page 28 that the "nothing" function "has no equational definition since its domain is empty"; surely, the equation "nothing $= \emptyset$" defines it!

In Chapter 3, the generalization of the product construct on domains to the infinitary case is postponed to an exercise in Chapter 6 (numbered 18, not 16 as stated in the book), because it raises "technical problems". There should also be some words here on the infinitary generalization of the disjoint sum construct, used in defining D^*.

In Chapter 5, the programming language syntax is not introduced until page 75, but is mentioned already on page 73; the word "loop" is used in the paragraph but "diverge" in the formal syntax. Chapter 5 also discusses some "nontraditional" treatments of stores; I found some of this counterintuitive, for instance on page 94, where there is an example of a divergent program which nevertheless gives a "result" *after* beginning to diverge; this suggests to me that an inappropriate semantics has been given.

The terminology for functionals in Chapter 6 is inconsistent: on page 107, F is "called a functional because it takes a function as an argument and returns one as a result"; but on page 113, a functional "is a continuous function" from D to D where, "usually, D is a domain of form $A \to B$, but it doesn't have to be." This chapter also mentions Hoare's logic for while-programs and sketches a proof of soundness with respect to the given semantic model. The proof (on page 127) for the sequential composition rule assumes strictness of the semantics. However, strictness is only stated on page 131 (and even then left as an exercise).

In Chapter 8, the type of the semantic function T′ is wrong, as it omits the type of its x-argument.

The example program on page 210 in Chapter 9, intended to illustrate the use of non-sequential control flow, in fact uses jumps only trivially: it would have the same (sequential) flow of control even if the jumps were removed. An obvious improvement would be to analyse the conventional label-and-jump implementation of while-loops. The example PROLOG-like language is given only a very cursory treatment, and would benefit from more detail, an example program, and a list of references to existing work on PROLOG. The remark on page 204 about the violation of sequentiality is obscure and it is too much to expect the reader to find an alternative semantics that is sequential.

There is some syntactic confusion in the language definition of Figure 9.5. Conditionals and while-loops appear with a test E in the figure, which switches to B in the semantic equations on pages 211 and 212; but B is now a meta-variable not for boolean-valued expressions but for blocks. It would also have been helpful to provide more details on proving the equivalence of the two definitions of while-loop semantics; this would have been a good illustration of fixed point techniques in action.

In Chapter 10 the treatment is sometimes thin. For instance, the chapter suffers from a plethora of acronyms (GRAM, DSL, LAMB, SPS, PSP, etc.); other than their presumably mnemonic qualities, the uninitiated reader would probably appreciate more explicit help. For instance, the relationships between LAMB, DLS and sugared lambda-notations are only implicit and the description of Mosses's SIS assumes that the reader is familiar with "the usual textual substitution method" (β-conversion). Since this is only supposed to be a brief survey this is probably a minor point. A more significant issue concerns the details and proof techniques required for the correctness of an operational semantics; it would have been helpful to have supplied a worked analysis of, say, the correctness of the stack-based VEC machine implementation. Another minor point concerns the use on page 233 of the notation $\prod_{i:Identifier} Nat$ for a dependent product, where surely it would have been simpler and sufficient to use the function space $Ide \rightarrow Nat$.

Chapter 11 on the solution of recursive domain equations by Scott's inverse limit construction includes worked examples on linear lists, procedures taking procedures as parameters, and recursive record structures. In addition to going through the construction of the domains it would have been useful to give details of the semantics of some simple program terms whose denotations belong to recursively defined domains, a prime candidate being a self-application procedure. Without such examples the reader is left with a lot of technical detail but little feeling for how the mathematics works in practice. It would also have been helpful to discuss the relationship between the recursively defined domain of linear lists and the D^* built from an infinite sum and used for finite lists earlier in the book.

In Chapter 12 the semantic domain of resumptions used to model concurrency is recursively defined, and I was surprised that the author did not state this explicitly

and did not relate this material to that of the previous chapter. The discussion of powerdomains is a bit careless in places. Since the empty set does not belong to the Egli-Milner powerdomain it is confusing to say (page 293) that "the constant $\emptyset$ represents the set $\{\perp\}$". The motivation for the inclusion of $\perp$ in infinite subsets is best set up in the framework of finitely branching computation trees, appealing to König's lemma (as on page 298), rather than by informal remarks like "if a computation has an infinite set of possible results, it will have to run forever to cover all the possibilities, hence nontermination is also a viable result" (again, page 293). Similar remarks apply to the Smyth powerdomain, where we read that "$\emptyset$ represents D"; again, the empty set does not belong to the powerdomain. I do not see why the author singles out the Egli-Milner powerdomain as being "useful for analyzing the operational properties of a language". The other powerdomains are also useful, each focussing on different aspects of program behaviour.

Some of the exercises contain errors, mostly minor but still annoying. There is a typographical mistake in the definition of Ackermann's function (exercise 2, page 132). More seriously, in exercise 10 on page 18 the phrase "transfinite induction" is used for what is usually known as "course-of-values" induction or "complete" induction of the natural numbers. Moreover, since the principles of course-of-values induction and mathematical induction on the natural numbers are interderivable in first order logic, i.e. any property provable by course-of-values induction is deducible from a property provable by mathematical induction, and vice versa, it does not make sense for the exercise to ask for a property that is "provable by transfinite induction and not by mathematical induction". Presumably the author means to find an example which cannot be proved *directly* but requires reformulation to fit the induction schema.

My final technical remark concerns a statement of apparent importance, made almost peripherally in an exercise (page 173). "The function notation used for denotational semantics definitions has its limitations, and one of them is its inability to simply express the Pascal-style hierarchy of data type". I think this conveys an erroneous overly pessimistic impression. Firstly, Pascal actually suffers from a restriction in the types allowed to parameters of procedures, so that in fact it is not possible in Pascal to define procedures of arbitrarily high order; it is more common to refer to the ALGOL-style hierarchy of types, in which procedures may take procedures as parameters. Secondly, it *is* possible to use denotational methods even for languages with an infinite number of types (which is presumably the cause for concern here). There are indeed existing semantic treatments of lambda-calculus (typed and "untyped") and ALGOL-like languages which encompass an infinite hierarchy of types; a typical approach might involve type-indexed families of semantic functions. Perhaps the key word in the quoted sentence is intended to be "simply". It would have been fairer to say that there are still issues to be settled in determining simple and elegant semantics for programming languages with higher types, and to list problematic features specific to this setting. Again, a final survey chapter would have been a good place for such a discussion, and it is a pity that Schmidt did not round off his book that way.

Summary

Overall, the book offers a well-intentioned attempt to broaden the appeal of denotational semantics to a wider audience. Schmidt has done a good job in trying to make the material accessible, while also including mathematical details. Some stylistic decisions are unfortunate and render the text less clear in places. The incorporation of material on inverse limit solutions to recursive domain equations, powerdomains, non-determinism and concurrency, and the generally more up-do-date nature of the material, distinguish this book from the competing literature. The author has been careful to warn the reader that certain topics (such as full abstraction) are still the subject of active research, and indeed because of this he only touches briefly on such issues in the text. On the whole, despite notational drawbacks and a few more or less minor problems, I think that the book may be recommended as a basis for a course on semantics. There is more than enough material for a single semester course, but Schmidt makes reasonable suggestions on course planning in his preface. To provide a well rounded comprehensive coverage of the material and its application to programming language design and implementation I feel it would be advisable in addition to consult Tennent's book, and possible Stoy or Gordon.

Steve BROOKES
CMU,
Pittsburg, U.S.A.

Brains, Machines and Mathematics. By M.A. Arbib. Springer, Berlin/Heidelberg/New York/London/Paris/Tokyo, 1987, Price DM 55.00, ISBN 0 3 540 96539 4.

Near the end of his life, John von Neumann produced a short book [1] in which he argued that the human central nervous system must employ a kind of mathematics which is structurally essentially different from the mathematics that we consciously and explicitly work with. In other words: the way our brains process most of their information is not at all like the way a mathematician solves algebraic equations or proves theorems on a blackboard. If so—and subsequent experience has done nothing to cast doubt on von Neumann's conclusion—then we need a mathematical analysis of information processing appropriate to biological organisms. It is with the quest for such a *biological mathematics* that Michael Arbib's book, *Brains, Machines and Mathematics*, is avowedly pre-occupied.

Arbib's point of departure is what he sees as a confluence of several intellectual currents concerned with the notions of intelligent automata and control. These currents are:

(i) Craik's theory of reasoning-by-simulation [2],
(ii) Rosenbleuth, Wiener and Bigelow's study of feedback and control [3] and
(iii) McCulloch and Pitts' threshold units [4],